Chapter 5. Describing a SOAP Service

Having seen the basic steps in implementing web services, you're now ready to explore technologies that make it easier to use web services that have already been deployed. Specifically, this chapter focuses on the Web Service Description Language (WSDL), which makes possible automated code-generation tools to simplify building clients for existing web services. WSDL also forms an integral component of the discovery process we'll see in Chapter 6.

5.1 Describing Web Services

The introduction of web services in Chapter 1 mentioned that one of the key things that sets web services apart from other types of applications is that they can be made self-describing. Here, we describe what that means.

Every application exposes some type of functionality; you invoke that functionality through various types of operations. Those operations require you to provide specific pieces of information. Once the operation is complete, the application may return information back to you. This entire exchange must be conducted using some agreed upon protocol for packaging the information and sending it back and forth. However, most applications typically require you, the developer, to describe how all of this is supposed to happen. The specific details of how a service is implemented become entrenched in the application. If any changes need to be made, the application must be changed and recompiled. These applications are not very flexible.

With web services, though, it is possible to allow applications to discover all of this information dynamically while the application is being run. This ability makes changes easier to accommodate and much less disruptive.

The SOAP specification does not address description. The de facto standard specification used to make web services self-describing is the Web Services Description Language (WSDL). Using WSDL, a web service can describe everything about what it does, how it does it, and how consumers of that web service can go about using it.

There are several advantages to using WSDL:

1. WSDL makes it easier to write and maintain services by providing a more structured approach to defining web service interfaces.
2. WSDL makes it easier to consume web services by reducing the amount of code (and potential errors) that a client application must implement.
3. WSDL makes it easier to implement changes that will be less likely to "break" SOAP client applications. Dynamic discovery of WSDL descriptions allows such changes to be pushed down automatically to clients using WSDL so that potentially expensive modifications to the client code don't have to be made every time a change occurs.

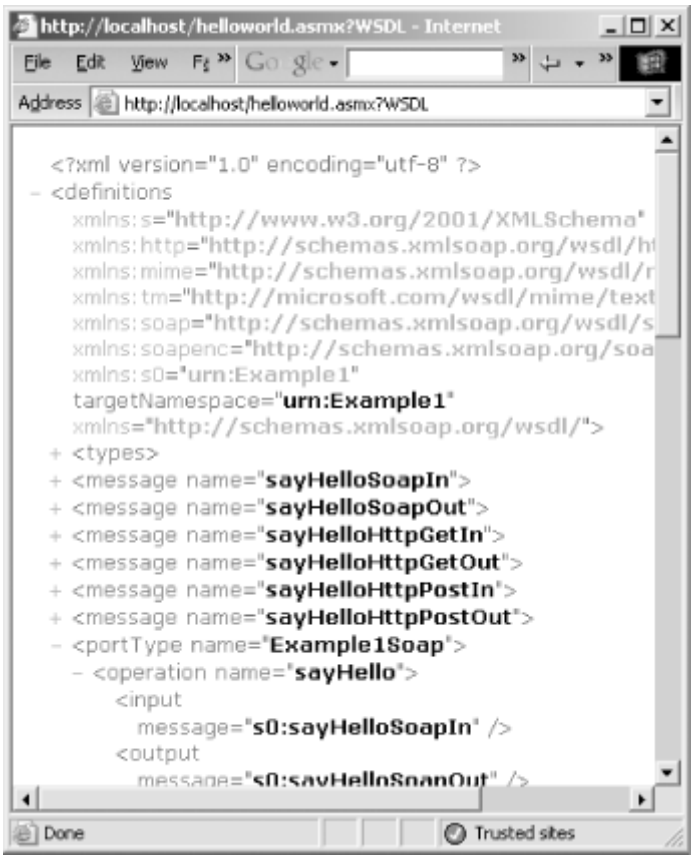
WSDL is not perfect, however. Currently, there is no support for versioning of WSDL descriptions, so web services providers and consumers need to be aware that when significant changes to a WSDL description occur, there may very well be problems propagated down to

the client. For the most part, however, WSDL descriptions should be treated in a similar manner to traditional object interfaces—where the definition of the service, once put into production, is immutable and cannot be changed.

Another key point is that, for the most part, web service developers will not be required to manually create WSDL descriptions of their services. Many toolkits include tools for generating WSDL automatically from existing application components.

Microsoft's .NET platform, for example, will automatically generate a WSDL description of deployed *.asmx* services simply by appending *?WSDL* to the URL of the *.asmx* file. If you have .NET and the *HelloWorld.asmx* service from Chapter 3, open your web browser and append the *?WSDL* to the end of the service's URL. You will see a dynamically generated WSDL description of the Hello World service, shown in Figure 5-1.

Figure 5-1. Automatically generated WSDL description for the .NET Hello World service



Keep in mind that not every web services toolkit includes WSDL support; third party add-ons may be required. IBM supplies an extension to Apache SOAP called the Web Services ToolKit that provides comprehensive WSDL support on top of Apache SOAP. WSIF, another IBM tool that we will take a look at in just a minute, is another example of a WSDL-enabling add-on for Apache SOAP. Apache Axis, when complete, will include built-in support for the use and creation of WSDL documents.

Although you can, and many do, use SOAP without WSDL, WSDL descriptions of your services make life easier for consumers of those services.

5.1.1 A Quick Example

To demonstrate quickly the difference that using a WSDL description of a web service can make in terms of the amount of code necessary to access a web service from Java, let's create a WSDL description for the Hello World web service and use the IBM Web Service Invocation Framework (WSIF) tools to invoke it. WSIF is a Java package that provides a WSDL-aware layer on top of Apache SOAP, allowing us to call SOAP services easily given only a WSDL description. It can be downloaded from http://alphaworks.ibm.com/tech/wsif. Within this service description, we will point to the Perl-based Hello World service created in Chapter 3.

The WSDL file begins with a preamble, then defines some messages that will be exchanged. This preamble is shown in Example 5-1.

Example 5-1. WSDL preamble

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorldDescription"
   targetNamespace="urn:HelloWorld"
   xmlns:tns="urn:HelloWorld"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:message name="sayHello_IN">
    <part name="name" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="sayHello_Out">
    <part name="greeting" type="xsd:string" />
  </wsdl:message>
```

Next, the WSDL defines how a method translates into messages. This is shown in Example 5-2.

Example 5-2. WSDL showing how a method corresponds to messages

```
  <wsdl:portType name="HelloWorldInterface">
    <wsdl:operation name="sayHello">
      <wsdl:input message="tns:sayHello_IN" />
      <wsdl:output message="tns:sayHello_OUT" />
    </wsdl:operation>
  </wsdl:portType>
```

Then the WSDL defines how the method is implemented (see Example 5-3).

Example 5-3. WSDL showing the implementation of the method

```
  <wsdl:binding name="HelloWorldBinding"
                type="tns:HelloWorldInterface">
    <soap:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http"
     />
    <wsdl:operation name="sayHello">
      <soap:operation soapAction="urn:Hello" />
```

```
      <wsdl:input>
        <soap:body use="encoded"
                   namespace="urn:Hello"
                   encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
        />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="encoded"
                   namespace="urn:Hello"
                   encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
        />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
```

And finally the WSDL says where the service is hosted (Example 5-4).

Example 5-4. WSDL showing the location of the service

```
  <wsdl:service name="HelloWorldService">
    <wsdl:port name="HelloWorldPort"
               binding="tns:HelloWorldBinding">
      <!-- location of the Perl Hello World Service -->
      <soap:address
        location="http://localhost:8080" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

The values of the `name` attributes in WSDL (e.g., `HelloWorldInterface` and `HelloWorldBinding`) are completely arbitrary. There are no defined naming conventions you should follow.

The complete WSDL document, shown in full in Appendix C, would be placed either in a well-known or, as we will explain Chapter 6, a discoverable location on your web server so that it may be retrieved using a simple HTTP-GET request. Once that is done, we can invoke the WSIF `DynamicInvoker` class to invoke the web service. This can be done using a single command-line operation:

```
C:\book>java clients.DynamicInvoker http://localhost/sayhello.wsdl sayHello
James
```

Which will produce the output:

```
Hello James
```

This is a big difference compared to the code we used in Chapter 3 to invoke the exact same service. The WSDL description allowed the WSIF tools to automatically figure out what needed to be done with the Apache SOAP tools in order to send the message and process the results, and you didn't have to write a single line of code. While this is a fairly simple example (you won't be able to use a single command line for every web service that uses WSDL and WSIF, as we will demonstrate later), it does stress the point: we use WSDL because it makes it easier to write web services.

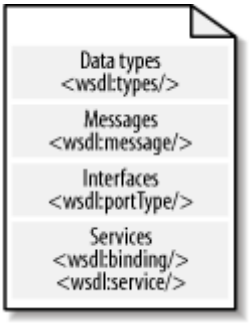
5.2 Anatomy of a Service Description

A web service description describes the abstract interface through which a service consumer communicates with a service provider, as well as the specific details of how a given web service has implemented that interface. It does so by defining four types of things: data, messages, interfaces, and services.

A *service* (`HelloWorldService` in our example) is a collection of *ports* (addresses implementing the service; see `HelloWorldPort` in the example). A port has both an abstract definition (the *port type)* and a concrete definition (the *binding*). Port types function as the specification of the software interface (`HelloWorldInterface` in this example), and are composed of collections of *operations* (the individual method signatures) that define the ordered exchanges of *messages* (`sayHello_IN` and `sayHello_OUT` in the example). Bindings say which protocols are used by the port, including the packaging protocol (SOAP in this case). A message is a logical collection of named *parts* (data values) of a particular type. The type of part is defined using some standard data typing mechanism such as the XML Schema specification.

The structure of a web service description is illustrated in Figure 5-2.

Figure 5-2. A service description describes four basic things about a web service: the data types, the messages, the interfaces, and the services



5.3 Defining Data Types and Structures with XML Schemas

Interoperability between applications on various operating system platforms and programming languages is most often hindered because one system's "integer" may not be exactly the same as another system's "integer." Because different operating systems and programming languages have different definitions of what particular base (or primitive) data types are not only called, but also how they are expressed when sent out over the wire, those operating systems and programming languages cannot communicate with each other.

To allow seamless cross-platform interoperability, there must be a mechanism by which the service consumer and the service provider agree to a common set of types and the textual representation of the data stored in them. The web services description provides the framework through which the common data types may be defined.

In WSDL, the primary method of defining these shared data types is the W3C's XML Schema specification. WSDL is, however, capable of using any mechanism to define data types, and may actually leverage the type definition mechanisms of existing programming languages or

data interchange standards. No matter what type definition mechanism is used, both the service consumer and the service provider must agree to it or the service description is useless. That is why the authors of the WSDL specification chose to use XML Schemas—they are completely platform neutral.

If you're unfamiliar with the XML Schema data representation system, now would be a good time to read the quick introduction in Appendix B.

Interestingly, while XML Schemas are used to define the data types, the message that is actually sent does not have to be serialized as XML. For example, if we decide to use a standard HTML form to invoke a web service, the input message will not be in XML syntax. The XML Schema specification itself recognizes that a schema may be used to describe data that is not serialized as an XML document instance, as evidenced by Section 2 of the XML Schema specification primer (http://www.w3.org/TR/xmlschema-0/):

The purpose of a schema is to define a class of XML documents, and so the term "instance document" is often used to describe an XML document that conforms to a particular schema. In fact, neither instances nor schemas need to exist as documents per se—they may exist as streams of bytes sent between applications, as fields in a database record, or as collections of XML Infoset "Information Items." —XML Schema Part 0: Primer, Section 2

So, if the data can be expressed as XML, regardless of whether it actually *is* expressed as XML, then XML Schemas can be used to describe the rules that define the data.

5.3.1 Using XML Schemas in WSDL

Once the data types are defined, they must be referenced within a WSDL description. Do so either by embedding the schema directly within the `<wsdl:types />` element, or by importing the schema using the `<wsdl:import />` element. While both approaches are valid, many WSDL-enabled tools do not yet properly support `<wsdl:import />`. The `<wsdl:types />` method is by far the most common. Examples of both approaches are shown here.

With `import`, you must declare the namespace that the XML Schema defines, then import the XML Schema document. This is shown in Example 5-5.

Example 5-5. Using import to reference a type definition

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorldDescription"
   targetNamespace="urn:HelloWorld"
   xmlns:tns="urn:HelloWorld"
   xmlns:types="urn:MyDataTypes"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

   <wsdl:import namespace="urn:MyDataTypes"
                location="telephonenumber.xsd" />

</wsdl:definitions>
```

Example 5-6 is the same definition but with the XML Schema embedded directly into the WSDL description.

Example 5-6. Embedding XML Schema directly to define types

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorldDescription"
   targetNamespace="urn:HelloWorld"
   xmlns:tns="urn:HelloWorld"
   xmlns:types="urn:MyDataTypes"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

   <wsdl:types>
     <xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
                 targetNamespace="urn:MyDataTypes"
                 elementFormDefault="qualified">
       <xsd:complexType name="telephoneNumberEx">
        <xsd:complexContent>
         <xsd:restriction base="telephoneNumber">
          <xsd:sequence>
            <xsd:element name="countryCode">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d{2}"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="area">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d{3}"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="exchange">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d{3}"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="number">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d{4}"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
          </xsd:sequence>
         </xsd:restriction>
        </xsd:complexContent>
       </xsd:complexType>
      </xsd:schema>
   </wsdl:types>

</wsdl:definitions>
```

5.4 Describing the Web Service Interface

Web service interfaces are generally no different from interfaces defined in object-oriented languages. There are input messages (the set of parameters passed into the operation), output messages (the set of values returned from the operation), and fault messages (the set of error

conditions that may arise while the operation is being invoked). In WSDL, a web service interface is known as a *port type*.

With this in mind, let's look again at the WSDL that we used earlier to describe the Hello World service. The relevant parts are shown in Example 5-7.

Example 5-7. Describing the Hello World service

```
<definitions ...>
  <wsdl:message name="sayHello_IN">
    <part name="name" type="xsd:string" />
  </wsdl:message>

  <wsdl:message name="sayHello_Out">
    <part name="greeting" type="xsd:string" />
  </wsdl:message>

  <wsdl:portType name="HelloWorldInterface">
    <wsdl:operation name="sayHello">
      <wsdl:input message="tns:sayHello_IN" />
      <wsdl:output message="tns:sayHello_OUT" />
    </wsdl:operation>
  </wsdl:portType>
  ...
</definitions>
```

The `portType` element defines the interface to the Hello World service. This interface consists of a single operation that has both an input and an expected output. The input is a message of type `sayHello_IN`, consisting of a single part called `name` of type `string`.

WSDL `portType`s do not support inheritance. It would be nice to be able to do something along the lines of Example 5-8, but it's not supported yet.

Example 5-8. Attempting inheritance with WSDL

```
<wsdl:definitions>
   <wsdl:portType name="HelloWorldInterface">
      <wsdl:operation name="sayHello" />
   </wsdl:portType>
   <wsdl:portType name="HelloWorldInterfaceEx"
          extends="HelloWorldInterface">
      <wsdl:operation name="sayGoodbye" />
   </wsdl:portType>
</wsdl:definitions>
```

The goal would be to have `SayHelloInterfaceEx` inherit the `sayHello` operation defined in `HelloWorldInterface`. You can't do that in WSDL right now, but support for some form of inheritance is being considered for future versions of the specification.

5.5 Describing the Web Service Implementation

WSDL can also describe the implementation of a given port type. This description is generally divided into two parts: the binding, which describes how an interface is bound to specific transport and messaging protocols (such as SOAP and HTTP), and the service, which

describes the specific network location (or locations) where an interface has been implemented.

5.5.1 Binding Web Service Interfaces

Just as in Java, COM, or any object-oriented language, interfaces must be implemented in order to be useful. In WSDL, the word for implementation is *binding* : the interfaces are bound to specific network and messaging protocols. In WSDL, this is represented by the `binding` element, shown in Example 5-9.

Example 5-9. Binding an interface to specific protocols

```
<wsdl:binding name="HelloWorldBinding"
              type="tns:HelloWorldInterface">

 <soap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>

 <wsdl:operation name="sayHello">
   <soap:operation soapAction="urn:Hello" />

    <wsdl:input>
      <soap:body use="encoded"
                 namespace="..."
                 encodingStyle="..." />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="encoded"
                 namespace="..."
                 encodingStyle="..." />
    </wsdl:output>
 </wsdl:operation>
</wsdl:binding>
```

This creates a new binding definition, representing a SOAP-over-HTTP implementation of the `HelloWorldInterface` port type. A SOAP-aware web services platform would use this information and the information contained in the port type and data type definitions to construct the appropriate SOAP envelopes for each operation.

The only difference between the `binding` element and the `portType` element is the addition of the `<soap:binding />`, `<soap:operation />`, and `<soap:body />` elements. These are the pieces that tell us how the messages are to be packaged. An instance of the input message for the `sayHello` operation bound to SOAP, using the earlier definition, would look something like Example 5-10.

Example 5-10. Instance of the message

```
<s:Envelope xmlns:s="...">
  <s:Body>
    <m:sayHello xmlns:m="urn:Hello">
      <name>John</name>
    </m:sayHello>
  </s:Body>
</s:Envelope>
```

The various `soap:` prefixed elements indicate exactly how the SOAP protocol is to be applied to the Hello World interface:

`<soap:binding />`

Defines the transport protocol and the style of the SOAP message. There are two styles: *RPC* and *document*. RPC indicates a SOAP message conforming to the SOAP RPC convention. Document indicates a SOAP messaging carrying some arbitrary package of XML data.

`<soap:operation />`

Defines the value of the `SOAPAction` header when the HTTP transport protocol is used.

`<soap:body />`

Specifies how the parts of the abstract WSDL message definition will appear in the body of the SOAP message by defining whether the parts are encoded (following the rules of some encoding style) or literal (arbitrary XML not necessarily following any defined set of encoding rules).

`<soap:fault />`

While not shown in the previous example, this element specifies the contents of the SOAP fault `detail` element. It works exactly like the `<soap:body />` element, defining how the `detail` part of the message will appear in the SOAP envelope.

`<soap:header />`

Specifies how parts of the message will appear in the header of the SOAP message.

`<soap:headerfault />`

Specifies how fault information pertaining to specific headers will appear in the header of the SOAP fault message returned to the sender.

`<soap:address />`

Specifies the network location where the SOAP web service has been deployed.

Alternatively, the binding could have specified a different packaging protocol for the messages—HTTP-GET, for instance. In this case, the binding element will include elements that describe how the message will appear within an HTTP URL. This is shown in Example 5-11.

Example 5-11. WSDL binding to HTTP-GET

```
<wsdl:binding name="HelloWorldBinding"
              type="tns:HelloWorldInterface">
 <http:binding verb="GET"/>
 <wsdl:operation name="sayHello">
    <http:operation location="sayHello" />
    <wsdl:input>
       <http:urlEncoded />
    </wsdl:input>
    <wsdl:output>
       <mime:content type="text/plain" />
    </wsdl:output>
 </wsdl:operation>
</wsdl:binding>
```

Each of the `http:` and `mime:` prefixed elements specify exactly how the port type is to be implemented. For example, the `<http:urlEncoded />` element indicates that all of the parts of the input message will appear as query string extensions to the service URL. An instance of this binding would appear as:

```
http://www.acme.com/sayHello?name=John
```

With the response message represented as nothing more than a stream of data with a MIME content type of `text/plain`.

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Type: text/plain;
Content-Length: 11

Hello James
```

5.5.2 Describing the Location of a Web Service

The final piece of information that a WSDL service implementation description must provide is the network location where the web service is implemented. This is done by linking a specific protocol binding to a specific network address in the WSDL service and port elements, as shown in Example 5-12.

Example 5-12. Linking a binding to a network address

```
<wsdl:service name="HelloWorldService">
  <wsdl:port name="HelloWorldPort"
             binding="tns:HelloWorldBinding">
    <soap:address location="http://localhost:8080" />
  </wsdl:port>
</wsdl:service>
```

In this example, we see that the Hello World service can be invoked through the use of SOAP messages, as defined by the `HelloWorldBinding` implemented at *http://localhost:8080*.

One interesting aspect of WSDL is that a service may define multiple ports, each of which may implement a different binding at a different network location. It is possible, for example,

to create a single WSDL service description for our three Hello World services written in Perl, Java, and .NET, as shown in Example 5-13.

Example 5-13. Multiple instances of the same server

```
<wsdl:service name="HelloWorldService">
  <wsdl:port name="HelloWorldPort_Perl"
             binding="tns:HelloWorldBinding">
     <soap:address location="http://localhost:8080" />
  </wsdl:port>
  <wsdl:port name="HelloWorldPort_Java"
             binding="tns:HelloWorldBinding">
     <soap:address location="http://localhost/soap/servlet/rpcrouter" />
  </wsdl:port>
  <wsdl:port name="HelloWorldPort_NET"
             binding="tns:HelloWorldBinding">
     <soap:address location="http://localhost/helloworld.asmx" />
  </wsdl:port>
</wsdl:service>
```

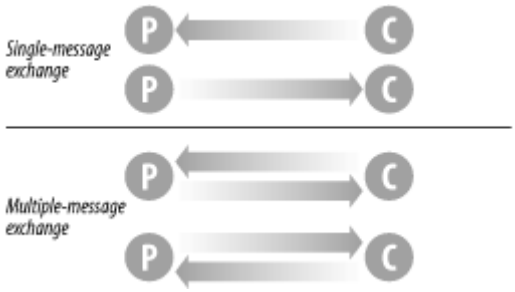
At this point the WSDL has described everything that a service consumer needs to know in order to invoke the Hello World web service we created in Chapter 3.

5.6 Understanding Messaging Patterns

A messaging pattern describes the sequence of messages exchanged between the service consumer and the service provider. The web services architecture supports two fundamental types of message patterns: single-message exchange and multiple-message exchange.

The definition of each pattern is based on whether the service provider or the service consumer is the first to initiate the exchange of messages, and whether there is an expected response to that initial message. Figure 5-3 illustrates two common message patterns.

Figure 5-3. Two patterns of message exchange between the service provider (P) and the service consumer (C)



Understanding these messaging patterns is an essential part of understanding how to build effective and useful web services.

5.6.1 Single-Message Exchange

A single-message exchange involves just that—a single message exchanged between the service consumer and the service provider. They are analogous to functions that do not have

return values. The message may originate with either the service provider or the service consumer.

To express a single-message exchange pattern in WSDL, define the abstract operation within the `portType` where the exchange will take place, as shown in Example 5-14.

Example 5-14. Single-message pattern in WSDL

```
<portType name="...">
  <operation name="Consumer_to_Provider">
    <input message="..." />
  </operation>
  <operation name="Provider_to_Consumer">
    <output message="..." />
  </operation>
</portType>
```

In WSDL, the `<input />` element is used to express the exchange of a message from the service consumer to the service provider. The `<output />` element is used to express the exchange of a message in the opposite direction, from the provider to the consumer.

5.6.2 Multiple-Message Exchange

And, obviously, multiple-message exchanges involve two or more messages between the service consumer and the service provider. These types of transactions range in complexity from simple function-style exchanges (calling a method on an object and returning a single result value), to a complex choreography of messages passed back and forth. The current version of WSDL, however, is only capable of expressing the simple function-style exchanges, as in Example 5-15.

Example 5-15. Function-style exchanges in WSDL

```
<portType name="...">
  <operation name="Consumer_to_Provider_to_Consumer">
    <input message="..." />
    <output message="..." />
  </operation>
  <operation name="Provider_to_Consumer_to_Provider">
    <output message="..." />
    <input message="..." />
  </operation>
</portType>
```

Again, all `<input />` messages originate with the service consumer and all `<output />` messages originate with the service provider.

5.6.3 Complex Multiple-Message Exchanges

By itself, WSDL is only capable of describing very rudimentary message exchange patterns. WSDL lacks the added ability to specify not only what messages to exchange in any given operation, but also the sequencing of operations themselves. Quite often, for example, it may be useful to specify that a service consumer must `login` before attempting to `deleteAllRecords`. WSDL has no way to describe such sequencing rules. A future version

of WSDL may allow such sequencing to be defined, either natively or through various extensibility mechanisms. Specifications such as IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG (pronounced "slang") have also been designed to deal with such sequencing issues from the point of view of a workflow process. These specifications will not be covered in this book.

5.6.4 Intermediaries

In Chapter 2 we discussed actors and message paths. A message path is the path a SOAP message takes on its way from the service consumer to the service requester. This path may be through several intermediary web services called actors, each of which may do something when it receives the message.

Intermediaries do not change the exchange pattern for a given operation. For example, a request-response operation between the service consumer and the service requester is still a request-response style operation. The only difference is that the request and the response messages may make a few additional stops on their way to their final destination. WSDL does not yet provide any facilitities for communicating the path that a message is to take.